The B Method as an Environment for the Verification of Eiffel Programs: A Case Study

Juan Oviedo and Nazareno Aguirre

Departamento de Computación, FCEFQyN,
Universidad Nacional de Río Cuarto,
Enlace Rutas 8 y 36 Km. 601, Río Cuarto, Córdoba, Argentina,
{joviedo, naguirre}@dc.exa.unrc.edu.ar

Abstract

In this paper we present an attempt to represent Eiffel programs as B specifications. Our purpose is to use the B method as an environment for the verification of the correctness of Eiffel programs.

We study the difficulties associated with the representation of object oriented features (inherent to Eiffel programs) in the non object oriented B language. We use an extension to B in order to represent dynamic creation and deletion of objects, and show how object interaction can be achieved by borrowing some ideas from software architectures.

The paper is centred on a simple and well known case study, the traditional object oriented implementation of generic lists.

Keywords: Object orientation, Program Verification, B Method.

1 Introduction

Object orientation is one of the most successful modern methodologies for the design and implementation of software systems. It constitutes an elegant and efficacious realisation of well established programming principles such as modularisation, information hiding and reuse. If used appropriately, object orientation can lead to the development of quality software.

Currently, there exist various object oriented development techniques, and a wide variety of object oriented programming languages. A particularly interesting object oriented programming language is Eiffel [9]. Eiffel was created by one of the leading researchers in object orientation, Bertrand Meyer, and relies on a well founded design concept, called *design by contract* [10]. In the context of object orientation, a contract is a statement that regulates the interaction between an object and its clients, and is expressed in terms of assertions. Typical examples of these assertions are preconditions and postconditions for methods, and class invariants, but other kinds of assertions are also possible.

Eiffel supports the definition of contracts by means of assertions, and provides mechanisms for capturing contract violation at run time, based mainly on

the use of exceptions. One can then realise if "something went wrong" during the execution of a program, and act appropriately by catching the thrown exceptions. However, Eiffel does not directly provide a mechanism for verifying that a program, or more precisely its constituent classes, satisfy their corresponding contracts. Verifying that a class satisfies its contract is a very important task, since it would ensure that, if the class is employed appropriately (i.e., within the "terms and conditions" of its contract), it will provide the required functionalities (i.e., it will fulfill its contract).

In order to verify that a class satisfies its corresponding contract, Bertand Meyer proposed attaching an *abstract model* to each class, and translate the class into a mathematically defined version; in order to prove that the class is correct, one needs to check that the mathematical version of the class is consistent with its abstract model. As indicated in [12], this might be done by translating Eiffel classes into specifications in the B language. The B formal specification language is a model based formalism for software specification. It has an associated method, the B Method [1], and commercial tool support, including proof assistance [4][5].

In this paper, we study the difficulties associated with the representation of object oriented features (inherent to Eiffel programs) in the non object oriented B language. The B language does not directly support some object oriented constructs, most notably dynamic creation and deletion of components, inheritance and communication through object references. We use a previously proposed extension to B in order to represent dynamic creation and deletion of objects, and show how object interaction can be achieved by borrowing some ideas from software architectures.

The paper is centred on a simple and well known case study, the traditional object oriented implementation of generic lists.

2 The B Method

The B language belongs to the class of the so called model based formal specification languages. It has been used in industry with some success, in a number of applications ranging from the development of control systems to smart cards. As all formal methods, the B method provides a formal language to describe systems, allowing for analysis and verification of certain system properties prior to implementation. An important characteristic of B is that it covers the whole development process, from specification to implementation.

B specifications are centred on the notion of *abstract machine*. Abstract machines are the units of modularisation of specifications in B, and resemble modules of non object oriented imperative languages. An abstract machine encapsulates data and behaviour. Data is represented in terms of *variables*, and behaviour in terms of *operations*. Consider, as a simple example, the abstract machine in Fig. 1. This is a simple machine, composed of a single variable, and two operations for setting and getting the value stored in this variable. This variable is "typed" by a set which is passed as a parameter of the machine.

As seen in this abstract machine, the types of the variables are specified within the invariant of the machine. The invariant might contain more complex statements, indicating which properties must be maintained throughout the life time of the abstract machine.

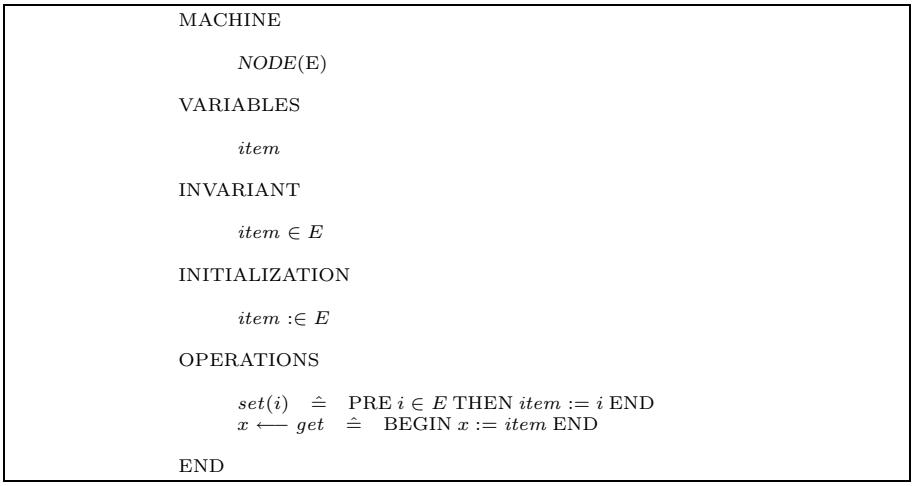

Figure 1: A simple abstract machine

Operations are specified by means of preconditions and multiple assignments. It is required that operations preserve the machine invariant. This constraint is part of the *proof obligations* of abstract machines. These constraints must be *discharged* in order to guarantee the consistency of the abstract machine.

2.1 Object Oriented Extensions to The B Method

Various facilities for structuring specifications are provided in B, helping to make the specification and refinement activities scalable. However, the B method has an important restriction regarding structuring mechanisms, namely, it does not provide dynamic creation and deletion of modules or components. All structuring mechanisms of B are *static*; they allow one to define abstract machines whose architectural structure in terms of other components is fixed, i.e., it does not change at run time [6]. Dynamic management of the population of components is a feature often associated with object oriented languages, since the replication of *objects* is intrinsic to these languages [11]. Indeed, dynamic management of "objects" appears frequently and naturally when modelling software, perhaps due to the success of object oriented methodologies and programming languages. However, fully fledged object oriented extensions of B would imply a significant change to B's (rather neat) syntax and semantics, and would excessively complicate the tool support implementation (especially in relation to proof support).

In [2], an extension of the syntax of B is proposed for supporting dynamic population management of components. This extension, in contrast with other proposed extensions to model based specification languages with this feature, such as some of the object oriented extensions to Z [14] or VDM [7], is not object oriented. Moreover, it does not imply any changes to the standard semantics of B, since it can be mapped into the standard language constructs [2]. The extension is essentially the provision of an extra structuring mechanism, the AGGREGATES clause, which intuitively allows us to dynamically "link" a set of abstract machines to a certain including machine. The semantics of AG-

GREGATES M relies on the generation of a *population manager* for M, i.e., a machine which represents a dynamic set of M, including operations for the creation and deletion of machine instances. A very important feature of the approach is that the generated population manager $M\,Manager$ for M is *correct by construction*, provided that M is correct [2]. Moreover, if one counts on a correct implementation for M, an implementation for $M\,Manager$ can be systematically constructed, which is also correct by construction [3].

The AGGREGATES M clause within a machine M' is then simply interpreted as INCLUDES M (or, more precisely, as EXTENDS M). As an example, consider the population manager generated for machine $NODE(E)$ is shown in Fig. 2. The meaning of AGGREGATES is graphically depicted in Fig. 3.

MACHINE

$NODEManager(E)$

VARIABLES

$NODESet,\ item$

INVARIANT

$(\forall n \cdot n \in NODESet \Rightarrow item(n) \in E) \wedge$
$(NODESet \subseteq \mathsf{NAME}) \wedge (item \in NODESet \rightarrow E)$

INITIALIZATION

$NODESet,\ item := \emptyset, \emptyset$

OPERATIONS

$add\_NODE(n) \ \widehat{=}$

PRE $n \in (\mathsf{NAME} - NODESet)$
THEN $NODESet := NODESet \cup \{n\}$ ||
$item :\in E$
END

$del\_NODE(n) \ \widehat{=}$

PRE $n \in NODESet$
THEN $NODESet := NODESet - \{n\}$ ||
$item := (dom(item) - \{n\}) <+ item$
END

$set(i, n) \ \widehat{=}$

PRE $i \in E \wedge n \in NODESet$
THEN $item := item \cup \{(n, i)\}$ END

$x \longleftarrow get(n) \ \widehat{=}$

PRE $n \in NODESet$
THEN $x := item(n)$ END

END

Figure 2: The NODEManager machine

3 Eiffel Programs

Eiffel is a strongly typed pure object oriented programming language. As such, its programs are written exclusively in terms of interrelated classes. The language itself supports the concepts of genericity, multiple inheritance and design

by contract; it also features a mechanism for exception handling, particularly related to contract violation, and provides automatic memory management.

Eiffel's syntax is based on languages such as Simula and Pascal. A simple example of the "flat form" of two Eiffel classes can be seen in Figs. 4 and 5. These two classes constitute (part of) a classical implementation of generic lists, using single linked nodes.

One can specify the state and the behaviour of objects by defining *features* (either attributes or routines) in their corresponding classes.

The language integrates the definition of assertions, such as preconditions and postconditions for "methods", variants and invariants for loop statements, and class invariants, directly in the source code. These assertions are useful for specifying and documenting classes, and can optionally be checked at run time. When an assertion is violated, an exception is thrown. This exception must be managed by some object in the invocation chain or will otherwise result in an error that aborts the program execution.

As we will see later on in this paper, we are not concerned, for the moment, with the implementation of methods. So, we have intentionally omitted, in the examples, the implementations of the methods.

Eiffel provides interesting features related to information hiding. As typical in other programming languages, one can define an "interface" for a class, indicating which features are "exported". But in contrast with other programming languages, in Eiffel one must specify which classes are allowed to access the exported features. The traditional public/private exporting becomes a special case of this more sophisticated notion of interface; in order to represent these, Eiffel provides two special built-in classes, ANY and NONE. A feature exported to ANY is "public", since ANY is an abstract class that is at the top of the Eiffel class hierarchy; class NONE, on the other hand, is at the bottom of the hierarchy and no class can inherit from it, so a feature exported to NONE is "private".

As we mentioned, Eiffel supports multiple inheritance, and the corresponding mechanisms to avoid problems related to it, such as repeated inheritance and name collisions. Contracts are inherited too; this is regulated by the "subcontracting principle", which establishes that a redefined method must preserve or weaken the precondition, and maintain or strengthen the postconditions.

Although we have been concerned with the description of Eiffel as a programming language, it is better understood if seen as a methodology for software

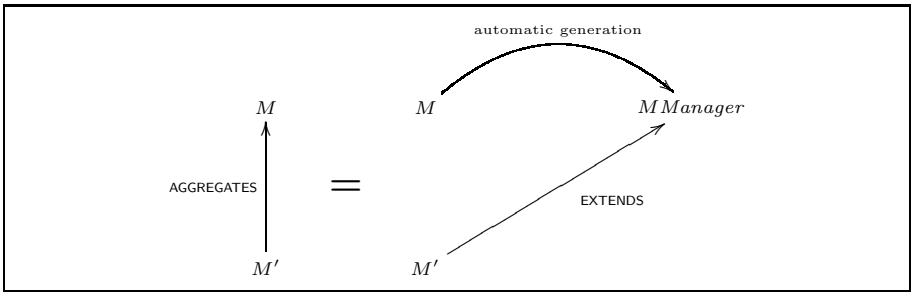

Figure 3: The meaning of AGGREGATES in terms of EXTENDS.

```
class LINK[E]
   creation make
   feature {NONE}
       item: E
       next: LINK[E]
   feature {ANY}
       make (i: E; n: LINK[E]) is
           ...
       ensure
           item = i
           next = n
       end

       set_item(i: E) is
          ...
       ensure
           item = i
       end

       set_next(n: LINK[E]) is
          ...
       ensure
           next = n
       end
       ...
end -- LINK[E]
```

Figure 4: Class Link

development. Eiffel, as a methodology, aims at good quality and productivity, by taking advantage of well established programming principles, and supporting them within the programming language. Eiffel also covers the entire development process; during the earliest stages, one can use Eiffel to directly specify the abstract properties of the objects by using assertions; in later stages, one can implement the software using Eiffel again, and test the programs against their specifications.

4 An Overview of the Approach

The reader might already find some similarities between Eiffel, as a method, and B. Both cover the entire development process, and share common notions such as class invariants (abstract machine invariants in B), preconditions and postconditions for methods (operations in abstract machines), etc. However, as we argue in this paper, their differences are not trivial to overcome. In particular, there exists an important "gap" when trying to map Eiffel programs into B specifications, namely the lack of object orientation in B.

Eiffel merges, in the same language, constructs for specifications in terms

```
class LIST[E]
 creation make
 feature {NONE}
     head: LINK[E]
 feature {ANY}
     make is
     ...
     ensure
         head = Void
     end

     insert_front(e: E) is
     require
         e /= Void
     ...
     ensure
         head.item = e
         head.next = old head
     end
     ...
end -- LINK[E]
```

Figure 5: Class List

of assertions and for implementation. B, on the other hand, has specialised and separated language constructs for specification and implementation. In B, one starts by specifying an abstract machine, say M. After proving that M is consistent (operations preserve the machine invariant, etc), one can define a refinement (resp. an implementation) for M. B counts on a number of conditions which must be discharged in order to prove that the proposed refinement (resp. implementation) is correct with respect to M. The tool support associated with the B method provides theorem proving facilities for carrying out this verification task.

Basically, the approach we propose in order to use the B method as an environment for the verification of Eiffel programs is the following:

> Given an Eiffel class P, with contracts expressed as assertions, two B specifications are synthesised:
>
> - an abstract machine M_P, which is constructed *only* from the assertions of P, and captures its invariant and pre and post-conditions of its operations,
> - an implementation I_P, constructed *only* from the code of P, and which represents the realisation of P's methods.
>
> Verifying the correctness of class P thus reduces to prove that I_P is a valid implementation of M_P. One can now attempt to prove this within B's mathematical framework, and by using the available tool support.

Graphically, this can be illustrated as in Fig. 6. The part of this diagram which is wrapped in dotted lines is what we study in this paper.

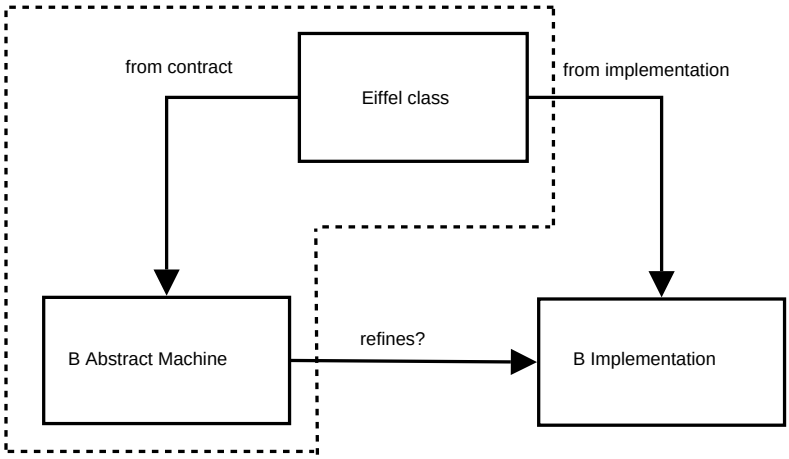

Figure 6: Graphical view of the proposed approach to the verification of Eiffel classes

5 A Case Study

In order to study the possible difficulties in representing Eiffel programs as B specifications, we choose to develop a case study. This case study consists of a simple and widely known datatype implementation, more precisely, a traditional object oriented implementation of generic lists. Since object *pointers* or *references* are particularly useful in object oriented implementations (e.g., they are normally used to implement class associations and client-provider relationships), we decided to base our study on the single linked reference based implementation of lists.

Single linked reference based lists can be implemented in Eiffel by defining a LINK class, meant to hold an item of the list, and having as a feature a reference to the link holding the "next item" in the list. An Eiffel definition of such a class was already shown in Fig. 4. Again, we have omitted the actual bodies of the methods of this class, and just showed their corresponding *contracts*, i.e., their pre and postconditions. Their implementations are not relevant to us for the moment, since we are only concerned, at this stage, with translating the "specification" of classes, i.e., their contracts. However, the implementations of these methods are obvious. Note that class LINK is *generic*, since it is parameterised by a (class) type E.

With class LINK defined, a single linked reference based list can be easily constructed, in the typical way, i.e., with a "reference" to the head of the list. The corresponding Eiffel class was previously shown in Fig. 5. Again, we have omitted the bodies of the methods of this class, but most readers should be already familiar with that.

Let us now start trying to define an appropriate translation of this Eiffel pro-

gram. We would prefer to maintain the architectural structure of the program, i.e., to provide one abstract machine for LINK, and another abstract machine for LIST, based on the first one. This is so due to the fact that, in specifications, modularisation is also important: modular specifications allow us to alleviate the proof efforts (for proving consistency, for instance) and help us understand complex specifications [4].

5.1 Representing class LINK

Class LINK should then be represented by an abstract machine LINK. Our first task in representing class LINK's constituents is the characterisation of its parameter E. This does not constitute any difficulty, since it can trivially be defined as a set parameter of the corresponding abstract machine. A different case is that of the representation of LINK's attributes (i.e., features that correspond to variables). The first of these is `item`, and its type is E. The second one, `next` is the problematic one, since its type is LINK, the very same class we are defining. In B, this is not allowed; abstract machines can only be hierarchically organised in terms of simpler machines, and no recursion or dynamism is permitted in this structure. This variable and its associated methods cannot be represented within LINK. LINK's representation as a B abstract machine becomes reduced to abstract machine *NODE*, given in Fig. 1. Since the "link" to the next item has been lost in the translation of LINK into a B abstract machine, we prefer to call the resulting abstract machine *NODE*, instead of LINK.

Of course, class *NODE* is not a faithful representation of class LINK, since a crucial part of it was lost in the translation. We will have to find a way of representing the "lost link"; as we will see, we will do this by borrowing some ideas from software architectures, externalising object interactions, and defining connectors.

5.2 Representing References

Since references to other modules or components is not, in general, straightforwardly supported in B, we need to represent them *outside* component definitions. A possible approach is that of *software architectures* [13]. In software architectures, systems are built out of components, which interact through *connectors*. Communication between components (objects in our case) is then *externalised* from the component definitions.

As we explained before, a special type NAME is used to characterise names of instances. In order to represent the 'next' attribute of the LINK original class, we can employ a binary relation *next* on NAME, relating instances of NODE within class LIST to their right neighbour. Since each instance of NODE has at most one right neighbour, relation *next* must be functional. But, of course, these elements should be defined within abstract machine LIST.

5.3 Representing class LIST

As is the case for class LINK, we represent class LIST as a separate abstract machine LIST. As we mentioned, attribute "next" originating in class LINK, has to be defined within this abstract machine. We also have a proper attribute of class LIST, namely variable *head*. The resulting abstract machine is shown in

Fig. 7. We make use of the AGGREGATES clause, that allows us to have several "dynamic instances" of the *NODE* abstract machine, as one expects to have in a list. Such instances are now elements of a set called NODESet, and controlled "dynamically" by a NODEManager abstract machine.

We assume the existence of a special constant Void which does not belong to NAME. Void is meant to represent, as in programming languages, a null reference. In our example, we employ Void as an initial value for attribute head.

```
MACHINE
    List(E)
AGGREGATES
    NODE(E)
VARIABLES
    head, next
INVARIANT
    head ∈ NAME ∪ {Void} ∧
    next ∈ NODESet → NODESet
INITIALIZATION
    head := Void
OPERATIONS
    set_next(x, y)  ≙
        PRE  x ∈ NODESet ∧ y ∈ NODESet
        THEN
            next(x) := y
        END
    insert_front(e)  ≙
        PRE  e ∈ E
        THEN
            ANY x WHERE (x ∈ NAME − NODESet) ∧
            (x ≠ Void)
            THEN
               add_NODE(x) || item := item ∪ {(x, e)} ||
               next := next ∪ {x, head} || head := x
            END
        END
END
```

Figure 7: Abstract machine *List(E)*.

6 Related Work

There is some evidence of the need of representing object oriented features in B, for a variety of reasons. The work of H. Treharne [16] is close to ours. Treharne tries to supplement an object oriented development (using the UML notation) with B. As opposed to the approach of [2], followed in this paper, the representation of object oriented features in B in Treharne's work is unstructured. As

we explained before, it is in our interest to produce structured B specifications from object oriented programs/specifications, since structure in specifications helps one in dealing with the complexity of large systems, alleviates the proof efforts, etc.

Another line of work related to B and object orientation is that of B. Tatibouet et al. [15], who propose translating B specifications into less formal notations, such as class diagrams or statecharts. Their work is less related to ours, since the direction of the translation is the opposite, and the objectives are rather unrelated (we intend to use B to verify programs, whereas Tatibouet et al.'s main objective is to provide more familiar "front-ends" of B for software engineers without mathematical background).

There exist object oriented alternatives to some model based formal methods, particularly *Object Z*, *Z++* and *VDM++*. It is not our intention to provide a "fully flavoured" object oriented extension of B. Instead, we try to incorporate only *some* object oriented features into B, without destroying B's methodology. The main advantages of our approach are the compatibility with the tool support available and existing work on B. K. Lano proposes to use these object oriented variants of model based formal methods for formal object oriented development [8].

7 Conclusions

We studied some difficulties associated with the representation of object oriented features in the non object oriented B specification language. The motivation for our work is the use of the B language (and method) as a verification framework for object oriented programs, particularly Eiffel programs.

We have chosen to use an available extension to B, presented in [2], to characterise dynamic creation and deletion of modules or components (abstract machines in B). We have shown by means of a case study that the work in [2] is insufficient, and that some simple and very common specification tasks could not be achieved using the extension 'as is'.

Other concepts intrinsic to object orientation, such as inheritance, were not considered in this paper. As work in progress, we are trying to extend the work in [2] to deal with the problems acknowledged here, and we are already studying possible representations of inheritance and associations in B.

References

[1] J.-R. Abrial, *The B-Book, Assigning Programs to Meanings*, Cambridge University Press, 1996.

[2] N. Aguirre, J. Bicarregui, T. Dimitrakos and T. Maibaum, *Towards Dynamic Population Management of Components in the B Method*, in Proceedings of the 3rd International Conference of B and Z Users ZB2003, Turku, Finland, LNCS, Springer-Verlag, June 2003.

[3] N. Aguirre, J. Bicarregui, L. Guzmán and T. Maibaum, *Implementing Dynamic Aggregations of Abstract Machines in the B Method*, to appear in

Proceedings of the International Conference on Formal Engineering Methods ICFEM 2004, Seattle, USA, LNCS, Springer-Verlag, 2004.

[4] *The B-Toolkit User's Manual*, version 3.2, B-Core (UK) Limited, 1996.

[5] Digilog, *Atelier B - Générateur d'Obligation de Preuve, Spécifications*, Technical Report, RATP SNCF INRETS, 1994.

[6] T. Dimitrakos, J. Bicarregui, B. Matthews and T. Maibaum, *Compositional Structuring in the B-Method: A Logical Viewpoint of the Static Context*, in Proceedings of the International Conference of B and Z Users ZB2000, York, United Kingdom, LNCS, Springer-Verlag, 2000.

[7] C. Jones, *Systematic Software Development Using VDM*, 2nd edition, Prentice Hall International,1990.

[8] K. Lano, *Formal Object-Oriented Development*. Formal Approaches to Computing and Information Technology, Springer, 1995.

[9] , B. Meyer, *Eiffel: The Language*, Second printing, Prentice-Hall, 1991.

[10] B. Meyer, *Applying 'Design by Contract'*, in IEEE Computer, Vol. 25, No. 10, 1992.

[11] B. Meyer, *Object-Oriented Software Construction*, Second Edition, Prentice-Hall International, 2000.

[12] B. Meyer, *A Fremework for Proving Contract-Equipped Classes*, in Proceedings of the 10th International Workshop on Abstract State Machines 2003– Advances in Theory and Applications, Taormina, Italy, Springer, 2003.

[13] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Pearson Education, 1996.

[14] M. Spivey, *The Z Notation: A Reference Manual*, 2nd edition, Prentice Hall International, 1992.

[15] B. Tatibouet, A. Hammad, and J.-C. Voisinet, *From an abstract B specification to UML class diagram*, in Proceedings of the 2nd IEEE International Symposium on Signal Processing and Information Technology (ISSPIT'2002), Marrakech, Maroc, 2002.

[16] H. Treharne, *Supplementing a UML Development Process with B*, in Proceedings of Formal Methods Europe FME 2002, Getting IT Right, Lecture Notes in Computer Science, Copenhagen, Denmark, 2002.